06-02552 *Principles of Programming Languages*
Spring Semester 2009-10

The University of Birmingham
School of Computer Science

Handout 2: Lambda Calculus Examples

In this handout, we look at several examples of lambda terms in order to provide a flavour of what is possible with the lambda calculus.

1 Notations

For convenience, we often give names to the lambda terms we examine. These names will be either written in bold (such as **name**) or underlines (such as name). Note that these names are not part of the lambda calculus itself. They are external. We also feel free to use these names in other lambda terms. In that situation, we should think of the names as standing in for the lambda terms that they name.

Even though the lambda calculus is untyped, a large majority of the lambda terms that we look at can be given types. In fact, looking at the types of the terms provides insight into the kind of functions these terms represent. So, wherever possible, we mention the types of the functions. We use capital letters A, B, ... to represent arbitrary types and the $\rightarrow$ symbol to represent function types. For example, $A \rightarrow B$ represents the type of functions from A to B, i.e., functions that given A-typed arguments, return B-typed results. We use a bracketing convention to parse type expressions with multiple $\rightarrow$ symbols: A type expression of the form $A_1 \rightarrow A_2 \rightarrow \cdots A_n \rightarrow B$ means $A_1 \rightarrow (A_2 \rightarrow \cdots (A_n \rightarrow B) \cdots)$. We say that the $\rightarrow$ operator *associates to the right*.

2 Examples

1. Identity. The lambda term:

$$\mathbf{id} = \lambda x.\, x$$

denotes the identity function, i.e., the function that simply returns its argument as its result. Indeed, by β-equivalence, $(\lambda x.\, x)\ M \equiv M$ for any term M.

If the identity function is given argument of type A, the result is again of type A. So, the identity function has type $A \rightarrow A$ for every type A. (In other words, it has infinite number of types, one for each instantiation of the symbolic type A: **int** $\rightarrow$ **int** and **bool** $\rightarrow$ **bool**, (**int** $\rightarrow$ **bool**) $\rightarrow$ (**int** $\rightarrow$ **bool**) are some example instances.)

2. Selection. The lambda term:

$$\mathbf{fst} = \lambda x.\, \lambda y.\, x$$

takes two arguments and returns the first argument as the result (ignoring the second argument). Notice that $(\lambda x.\, \lambda y.\, x)\ M\ N \equiv (\lambda y.\, M)\ N \equiv M$ by β-equivalence.

However, the manner in which the two arguments are provided to **fst** is typical of the lambda calculus higher-order character. The **fst** function is first given an argument, say of type A, and it returns a function. This (returned) function takes another argument, say of type B, and returns the original first argument (of type A). In other words, the type of **fst** is $A \rightarrow (B \rightarrow A)$.

Similarly, the lambda term

$$\mathbf{snd} = \lambda x.\, \lambda y.\, y$$

returns the second argument that it is given (ignoring its first argument). It has the type $A \rightarrow (B \rightarrow B)$.

3. Constant functions. The function $\lambda x.\, 0$ returns 0 no matter what argument we give it. It is a "constant function". Similarly, $\lambda x.\, 1$ is a constant function that returns 1.

We can define a lambda term to build such constant functions:

$$\mathbf{K} = \lambda x.\, \lambda y.\, x$$

Now, $\mathbf{K}\ 0$ is the constant function that returns 0. In general, $\mathbf{K}\ x$ is the constant function that returns x.

Note that **K** and **fst** are the same function, in fact the same lambda term. The only difference is in our view of them.

4. Application. The lambda term:

$$\textbf{apply} = \lambda f.\, \lambda x.\, f\ x$$

takes a function and a value as argument and applies the function to the argument.
To give a type to the function, notice that f is a function and it takes x as an argument. So, if x is of type A then f must be of type $A \to B$ for some B. So, the overall type of **apply** can be written as

$$(A \to B) \to A \to B$$

$A \to B$ is a possible type of f, A is the possible type of x, and B is the result type of **apply** which is the same as result type of f.
The lambda term:

$$\textbf{twice} = \lambda f.\, \lambda x.\, f\ (f\ x)$$

is similar to **apply** but applies the function f twice. It applies f to x obtaining a result, and applies f to this result once more. Its type is similar to that of **apply** but, since f is applied again to the *result* of f, the argument type and the result type of f should be the same, say A. So, the overall type of **twice** is $(A \to A) \to A \to A$.
Similarly, the lambda term

$$\textbf{thrice} = \lambda f.\, \lambda x.\, f\ (f\ (f\ x))$$

applies f thrice.

5. Function composition. If f is a function of type $A \to B$ and g is of type $B \to C$, mathematicians speak of their *composition* which is function denoted $g \circ f$ of type $A \to C$. Given an argument, $g \circ f$ first applies f to the argument and then applies g to the result of this application.
We define a lambda term that captures function composition:

$$\textbf{comp} = \lambda g.\, \lambda f.\, \lambda x.\, g\ (f\ x)$$

Now, **comp** $g\ f$ is the same as what mathematicians write as $g \circ f$.
The type of **comp** can be expressed as

$$(B \to C) \to (A \to B) \to (A \to C)$$

for any types A, B and C. You should verify that this is indeed the correct type for **comp**.
Note that **twice** f is equivalent **comp** $f\ f$.
Similarly, **thrice** f is equivalent to **comp** f (**comp** $f\ f$) as well as **comp** (**comp** $f\ f$) f.

6. Self application. Here is a lambda-term that appears strange from a traditional mathematical point of view:

$$\textbf{sa} = \lambda x.\, x\ x$$

This function takes an argument x, which is apparently a function. It applies the function to itself and returns whatever is the result.
What is strange is that x is a function that can take itself as an argument. Are there any such functions? Indeed, there are. **id** is clearly a function that can be applied to itself. Notice:

$$\textbf{id id} = (\lambda x.\, x)\ \textbf{id} = \textbf{id}$$

The **fst** and **snd** functions can also be applied to themselves:

$$\begin{aligned}\textbf{fst fst} &= (\lambda x.\, \lambda y.\, x)\ \textbf{fst} = \lambda y.\, \textbf{fst}\\ \textbf{snd snd} &= (\lambda x.\, \lambda y.\, y)\ \textbf{snd} = \lambda y.\, y = \textbf{id}\end{aligned}$$

For a more substantive example of self-application, consider applying the **twice** function to itself:

$$\begin{aligned}\textbf{twice twice} &= (\lambda f.\, \lambda x.\, f\ (f\ x))\ \textbf{twice}\\ &= \lambda x.\, \textbf{twice}\ (\textbf{twice}\ x)\\ &= \textbf{comp twice twice}\end{aligned}$$

You can calculate that **comp twice twice** is a quite normal function that creates a four-fold application of a given function.
What happens if we apply **sa** to itself?

$$\textbf{sa sa} = (\lambda x.\, x\ x)\ \textbf{sa} = \textbf{sa sa}$$

So, if we try to use β-reduction to find out what **sa sa** means, we get nowhere. This term corresponds to an "infinite loop" in lambda calculus. It is denoted by the symbol Ω.

7. The Y combinator. The following famous term is called the **Y** combinator.

$$\mathbf{Y} = \lambda t.\,(\lambda x.\,t\,(x\,x))\;(\lambda x.\,t\,(x\,x))$$

This term looks almost like the self-application of **sa**, but it is different in that it involves an additional function t in a subtle way. Consider an application $\mathbf{Y}\,t$ and let us see what we can learn about it using β-reduction:

$$\begin{array}{lcll} \mathbf{Y}\,t & = & (\lambda x.\,t\,(x\,x))\;(\lambda x.\,t\,(x\,x)) & \\ & = & t\,((\lambda x.\,t\,(x\,x))\;(\lambda x.\,t\,(x\,x))) & \text{by } \beta\text{-reduction} \\ & = & t\,(\mathbf{Y}\,t) & \text{noticing that the inner term is } \mathbf{Y}\,t \end{array}$$

So, $\mathbf{Y}\,t$ is equal to the function t applied to itself! One can use this to repeatedly unfold $\mathbf{Y}\,t$.

$$\mathbf{Y}\,t = t\,(\mathbf{Y}\,t) = t\,(t\,(\mathbf{Y}\,t)) = t\,(t\,(t\,(\mathbf{Y}\,t))) = \cdots$$

This might seem like another form of an infinite loop, but it is actually quite useful. In fact, it is used to encode recursive functions in the lambda calculus.

Consider the recursive definition of a function such as the factorial:

$$\begin{array}{llcl} \textbf{define} & \text{factorial} & = & \lambda n.\,\text{if}\;(=n\;1)\;1 \\ & & & \quad (*\;n\;(\text{factorial}\;(-\;n\;1))) \end{array}$$

On the surface, this is a circular definition and cannot be expressed in lambda calculus. To resolve the difficulty, we first treat the right hand side of the definition as a function of "factorial":

$$\begin{array}{llcl} \textbf{define} & \text{factorial} & = & \underline{T}\;\text{factorial} \\ \textbf{define} & \underline{T} & = & \lambda f.\,\lambda n.\,\text{if}\;(=n\;1)\;1 \\ & & & \quad (*\;n\;(f\;(-\;n\;1))) \end{array}$$

The definition of $\underline{T}$ is quite normal, but the first line is still a circular definition. However, this is exactly the kind of circularity that the **Y** combinator allows us to capture. The **Y** combinator satisfies the equality $\mathbf{Y}\,\underline{T} = \underline{T}\,(\mathbf{Y}\,T)$. So, we can just say that factorial is $\mathbf{Y}\,\underline{T}$ and we get what we want without any circular definitions.

Does this actually work? Here is a sample calculation:

$$\begin{array}{lcll} (\mathbf{Y}\,\underline{T})\,2 & = & \underline{T}\,(\mathbf{Y}\,\underline{T})\,2 & \\ & = & \text{if}\;(=2\;1)\;1\;(*\;2\;(\mathbf{Y}\,\underline{T}\;(-\;2\;1))) & \beta\text{-reduction} \\ & = & (*\;2\;(\mathbf{Y}\,\underline{T}\;1)) & \text{calculating arithmetic} \\ & = & (*\;2\;1) & \text{separate calculation} \\ & = & 2 & \\ (\mathbf{Y}\,\underline{T})\,1 & = & \underline{T}\,(\mathbf{Y}\,\underline{T})\,1 & \\ & = & \text{if}\;(=1\;1)\;1\;(*\;1\;(\mathbf{Y}\,\underline{T}\;(-\;1\;1))) & \beta\text{-reduction} \\ & = & 1 & \text{calculating arithmetic} \end{array}$$

Thus, in general, all recursive function definitions can be represented in the lambda calculus as applications of the **Y** combinator. This gives the lambda calculus the power of Turing machine computations.

8. Subtleties of self application. Even though self-application allows calculations using the laws of the lambda calculus, what it means conceptually is not at all clear. We can see some of the problems by just trying to give a type to **sa** $= \lambda x.\,x\,x$. Suppose the argument x is of type A. But, since x is being applied as a function to x, the type of x should be of the form $A \to \ldots$. How can x be of type A as well as $A \to \ldots$? Is there a type A such that $A = (A \to B)$? In traditional mathematics (set theory), there is no such type.

However, we have just seen that there are quite a few functions that can be applied to themselves. We have also seen that we can usefully encode recursive functions using self application. Calculations using the lambda calculus produce quite normal and sensible results. This explains why the lambda calculus has been called a "calculus". It is a system for doing calculations. However, it does not have meaning. So it was thought for a long time.

Finally, in 1960s, Dana Scott, then a Professor at Oxford University, and himself a former student of Alonzo Church, discovered a meaning for the lambda calculus. He formulated structures called "domains" which can be used to represent types (instead of traditional sets). In domains, there are indeed types A such that $A = (A \to B)$. This led to the development of an elegant theory of domains, which serves as the foundation for the mathematical meaning of programming languages. A survey article on *Domain Theory*, written by S. Abramsky and A. Jung appears in the *Handbook of Logic in Computer Science*, Oxford University Press, 1994.

Self application is used very fundamentally in implementing object-oriented programming languages. Suppose we have an object x with a method m. We might invoke this method by writing something like $x.m(y)$. However,

inside the method m, there would be references to keywords like "self" or "this" which are supposed to represent the object x itself. But how does m know what the object x is? One way of solving the problem is to translate the method m into a function m' that takes *two* arguments: in addition to the proper argument y, the object on which the method is being invoked. So, the definition of m' looks like:

$$m' = \lambda\text{self}.\, \lambda y. \ldots \text{the body of m} \ldots$$

The object x has a collection of such functions encoding the methods. The method call $x.m(y)$ is then translated as $x.m'(x)(y)$. This is a form of self application. The function m', which is a part of the structure x, is applied to the structure x itself. The meaning of such self application is explained in the article "Two semantic models of object-oriented languages" by S. Kamin and U. S. Reddy in the volume *Theoretical Aspects of Object-Oriented Programming*, MIT Press, 1994.